**Assignment 1 – Task 2**

The data Crossword data structures that I set up were fairly intuitive, and the parts that weren’t (such as an array of StringBuilders) were suggested in the assignment. Building the DLB Trie was manageable as well, thanks to the labs and the lecture notes. Building the data structures themselves, processing the dict8.txt file and the test board files, and writing the helper method isAvailable all was straight forward. A lot of my struggle was spent on how to properly develop my recursive calls, and how to set up the logic of confirming or rejecting valid words and/or prefixes. For this, I spent many nights at home, as well as several hours in the Computer Science Resource Center. They (specifically Andrew Speers, or maybe Andrew Jia) were excellent resources. Andrew helped me clear up a lot of extra, unnecessary code that I had, as well as coached me on good coding practices (such as having helper methods do one simple thing). Building code in this manner is not a skill that I am too familiar with, as up to this point, my highest level class is CS445, so I was able to get away with sloppy coding.

Despite Andrew’s help and my own sleuthing, I am still encountering an issue where my programs will not tolerate boards that have solid/filled squares. I couldn’t figure out where to look for/how to handle this situation. I know it should be part of the logic, i.e. if a string is a valid prefix but not a word (searchPrefix returned a 1), but the next space is solid, it should be rejected. However, I couldn’t figure out if that check should be implemented in my wordPrefix method, or inside of the recursive loops (recurseOne or recurseMany). Blank boards or boards that had set characters were processed correctly.

With respect to 3x3 boards, the DLB solution may have been faster, but using informal methods, both the DLB and MyDictionary attempts appeared to be roughly the same performance. As the board sizes grew, however, it was clear that the DLB Trie was the superior searching algorithm, even when only reaching the first solution.

Time to First Solution

|  |  |  |
| --- | --- | --- |
| **Board Size** | **MyDictionary** | **DLB** |
| **3x3** | Immediate | Immediate |
| **4x4** | ~10.5s | Immediate |
| **5x5** | ~8s | Immediate |
| **6x6** |  | ~5.25min |

These times were the only ones that I was able to obtain, due to limited time to deadline (my fault). However, the time to find the first solution for a 6x6 board, which is running as I type this paper, is at least on the order of minutes for DLB, and projected to be even longer for MyDictionary. This is on top of the fact that my runtimes were significantly longer that the provided reference runtimes.

The long run times for MyDictionary are due to it being based on a brute force approach. The brute force approach has a run time of O(n\*m) where n is the number of words in the dictionary used and m is the maximum string length (or board dimension). The DLB Trie significantly improves on this run time, with a search time that is O(m) per search, where m is the string length. Even with the improved search time, however, each time the board dimensions increase by one, the number of possible solutions increase as well. I think that a small improvement to performance could be made by reading the test board in first, acquiring the board dimensions, and then reading in the dictionary file and culling any words that are greater in length than the board dimensions. For larger boards, this may not make much of a difference, but if a helper method were implemented to determine the longest word possible on a board (perhaps by checking the longest row or column for consecutive not-solid spaces), we could also prune the dictionary that way.

\* Huge shout-out to my TA, Marcus, and the CSRC for all the help. Without them, I’d still be scratching my head while staring at my recursive methods.